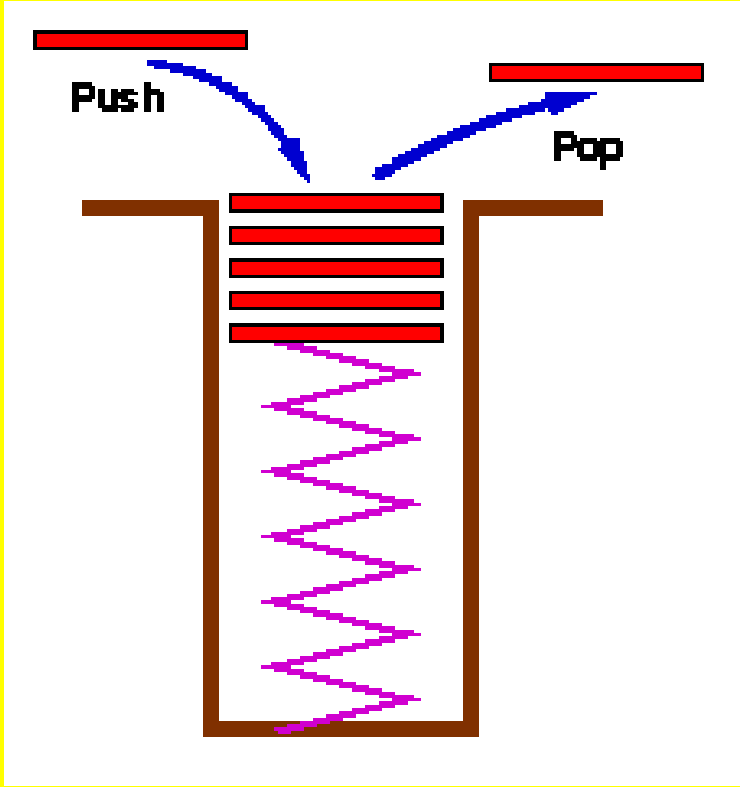
Lecture
Stack
Data Structures & Algorithms

- Syed Mudassar Alam

Stack

- Stack is a data structure that can be used to store data which can later be retrieved in the reverse or **last in first out (LIFO)** order.
- Stack is an ordered-list in which all the insertions and deletions are made at one end to maintain the LIFO order.
- It is also called Push Down list
- Operations are performed by one end called **TOP**


Push
Pop

Example of Stack

- Item in the Container
- Tennis balls in a box
- Books on the floor
- Gun Magazine

Stacks

- The operations defined on a stack are:
  1. Push - Store onto a stack
  2. Pop - retrieve from stack
  3. Top - examine the top element in the stack
  4. Is_empty - check if the stack is empty
  5. Is_Full - check if the stack is full
- A stack can be very easily implemented using arrays.
- Stack is implemented by maintaining a pointer to the top element in the stack. This pointer is called the *stack pointer.*

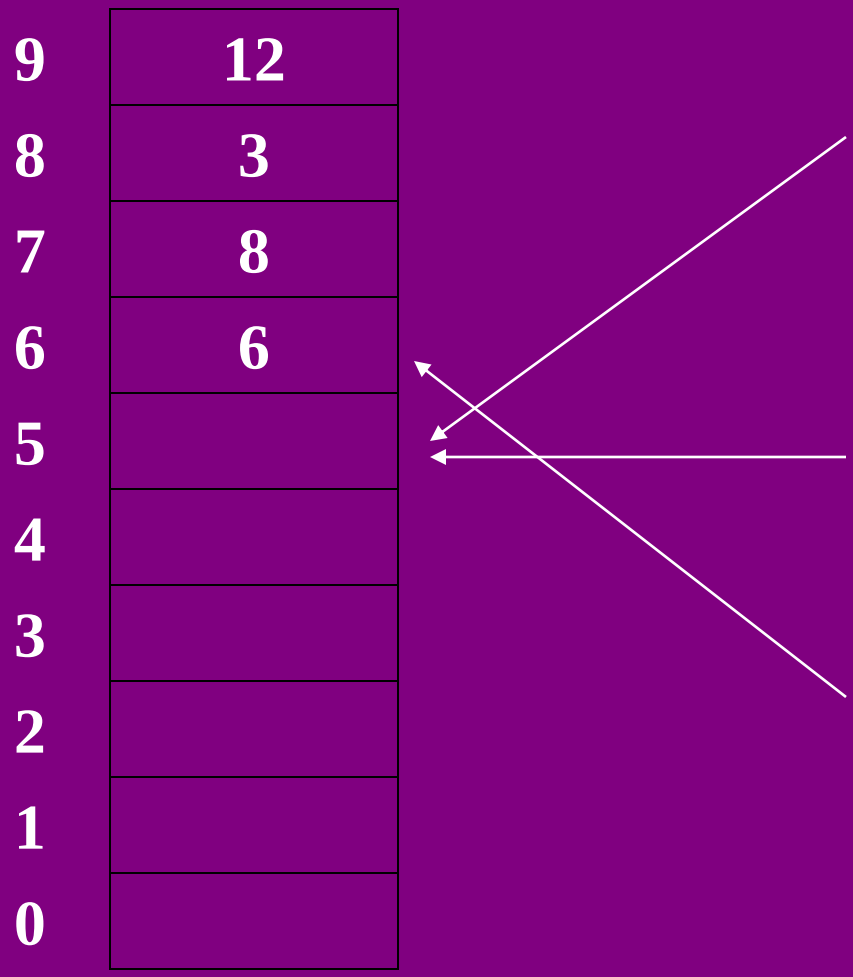
Stacks – Array Implementation

If a stack is implemented using arrays, the following two conventions can be used:

1. A stack can grow upwards, i.e., from index 0 to the maximum index, or it can grow downwards, i.e., from the maximum index to index 0.
2. *Stack pointer* can point to the last element inserted into the stack or it can point to the next available position.

Growing Downwards

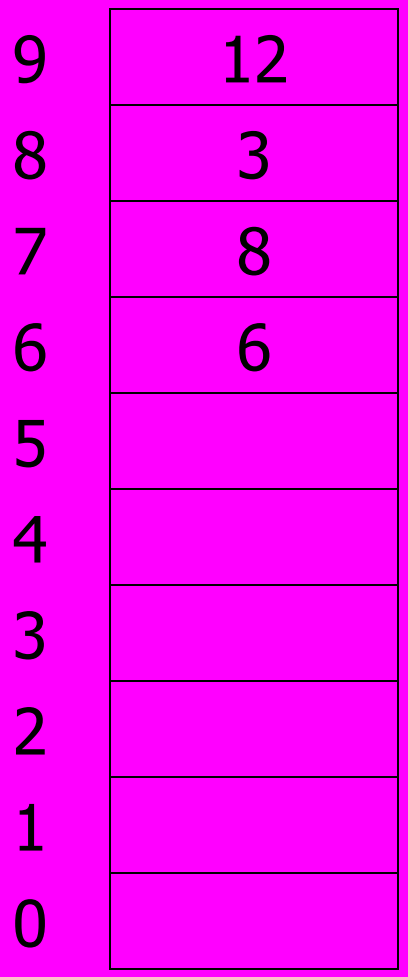
Initial state: stk_ptr = MAX - 1



- stk_ptr points to the next empty location
- Push – first add data to the stack, then decrement stk_ptr
- Pop – first increment stk_ptr and then take data out

Growing Downwards

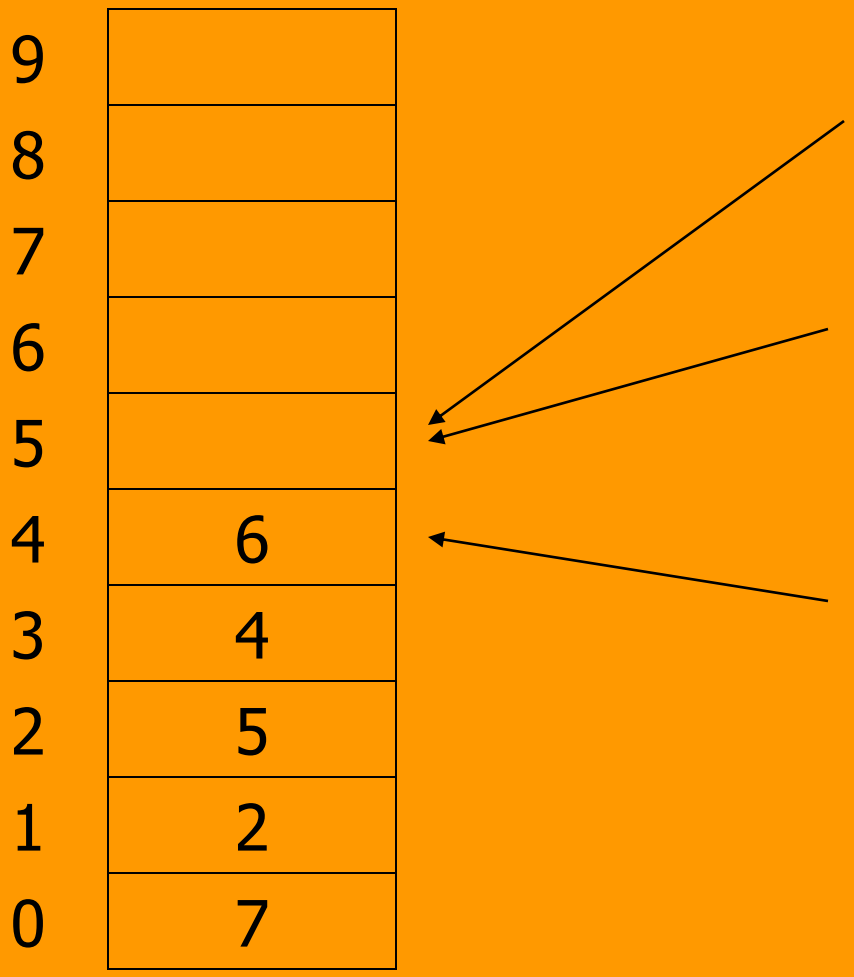
Initial state: stk_ptr = MAX



- stk_ptr points to the last element added to the stack
- Push – first decrement the stk_ptr and then add data
- Pop – first take data out and then increment stk_ptr

Growing Upwards

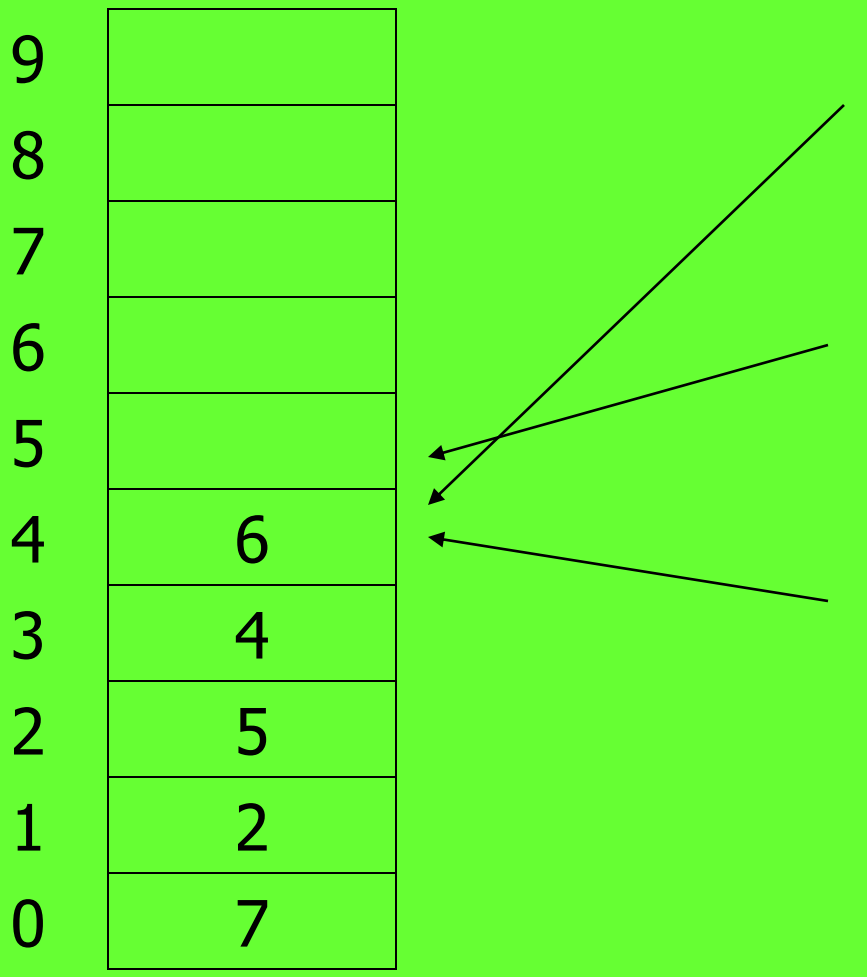
Initial state: stk_ptr = 0



- stk_ptr points to the next empty location
- Push – first add data to the stack then increment stk_ptr
- Pop – first decrement stk_ptr and then take data out

Growing Upwards

Initial state: stk_ptr = -1



- stk_ptr points to the last element added to the stack
- Push – first increment the stk_ptr and then add data
- Pop – first take data out and then decrement stk_ptr

Stacks – Array Implementation

```
class Stack {
private:
    int size;                          // maximum storage capacity
    int stk_ptr;                       // stack pointer
    int *stackArray;                   // array used to implement stack
public:
    Stack(int s );                     // constructor
    ~Stack() {delete [ ] stackArray; } // destructor
    bool push (int);                   // add an element to the stack
    bool pop(int &);                   // remove an element from stack
    bool isFull();                     // check if the stack is full
    bool isEmpty();                    // check if the stack is empty
};
```

```
Stack::Stack(int s)
{
    size = s;
    stk_ptr = 0;
    stackArray = new int[size];
}
```

```
bool Stack::isEmpty()
{
    return (stk_ptr == 0);
}
```

```
bool Stack::isFull()
{
    return (stk_ptr == size);
}
```

```
bool Stack::push(int n)
{
    if (! isFull() ) {
        stackArray[stk_ptr] = n;
        stak_ptr = stk_ptr + 1;
        return true;
    }
    else return false;
}
```

```
bool Stack::pop(int &n)
{
    if (! isEmpty() {
        stk_ptr = stk_ptr – 1;
        n = stackArray[stk_ptr];
        return true;
    }
    else return false;
}
```

Application of Stacks- Verifying parentheses are nested correctly

1. There are an equal number of right and left parentheses.
2. Every right parenthesis is preceded by a matching left parenthesis.

- Hence the following are incorrect:

((A+B) violates condition 1

)A+B(- violates condition 2

Algorithm

```
valid=true;
s=the empty stack;
while(we have not read the entire string){
read the next symbol (symb) of the string;
if(symb== ‘(‘ || symb== ‘[‘ || symb== ‘{‘ )
        Push symb on s;
if(symb== ‘)‘ || symb== ‘]‘ || symb== ‘}‘ )
            if(s is empty)
                    valid=false
            else{
                    pop element i from s
                    If(i is not the matching opener of symb)
                            valid=false;
            }
}
if(s is not empty)
            valid=false;

if(valid is true)
            String is valid
else
            String is invalid
```

Application of Stacks- "Evaluation of Expression"

- Evaluation of expression like

  a+b/c*(e-g)+h-f*i

  was a challenging task for compiler writers.
- It is a problem of parenthesization of the expression according to operator precedence rule.
- A fully parenthesized expression can be evaluated with the help of a stack.

Algorithm to Evaluate fully Parenthesized Expressions

1. while (not end of expression) do
    1. get next input symbol
    2. if input symbol is not “)”
        1. push it into the stack
    3. else
        1. repeat
            1. pop the symbol from the stack
        2. until you get “(“
        3. apply operators on the operands
        4. push the result back into stack
2. end while
3. the top of stack is the answer

Evaluation of Fully Parenthesized Expression

(a+(b/c))

Assuming a=2, b=6, c=3

Input Symbol	**Stack**	**Remarks**
(	(	Push
a	(a	push
+	(a+	push
(	(a+(	push
b	(a+(b	push
/	(a+(b/	push
c	(a+(b/c	Push
)	(a+2	Pop”(b/c” and evaluate and push the result back
)	4	Pop”(a+2” and evaluate and push the result back

Evaluation of Expressions

- The normal way of writing expressions i'.e., by placing a binary operator in-between its two operands, is called the *infix* notation.
- It is not easy to evaluate arithmetic and logic expressions written in infix notation since they must be evaluated according to operator precedence rules. E.g., a+b*c must be evaluated as (a+(b*c)) and not ((a+b)*c).
- The *postfix* or *Reverse Polish Notation* (RPN) is used by the compliers for expression evaluation.
- In RPN, each operator appears after the operands on which it is applied. This is a parenthesis-free notation.
- Stacks can be used to convert an expression from its infix form to RPN and then evaluate the expression.

INFIX and POSTFIX

Infix	Postfix
a+b*c	abc*+
a*b+c*d	ab*cd*+
(a+b)*(c+d)/e-f	ab+cd+*e/f-
a/b-c+d*e-a*c	ab/c-de*+ac*-
a+b/c*(e+g)+h-f*i	abc/eg+*+h+fi*-

Algorithm to Evaluate Expressions in RPN

1. while (not end of expression) do
    1. get next input symbol
    2. if input symbol is an operand then
        1. push it into the stack
    3. else if it is an operator then
        1. pop the operands from the stack
        2. apply operator on operands
        3. push the result back onto the stack
2. End while
3. the top of stack is answer.

Algorithm to Evaluate Expressions in RPN

(a+b)*(c+d) → ab+cd+*

Assuming a=2, b=6, c=3, d=-1

Input Symbol	Stack	Remarks
a	a	Push
b	a b	Push
+	8	Pop a and b from the stack, add, and push the result back
c	8 c	Push
d	8 c d	Push
+	8 2	Pop c and d from the stack, add, and push the result back
*	16	Pop 8 and 2 from the stack, multiply, and push the result back. Since this is end of the expression, hence it is the final result.

Algorithm for Infix to RPN Conversion

1. Initialize an empty stack of operators
2. While not end of expression do
   a. Get the next input token
   b. If token is
      i. "(" — push
      ii. ")" — pop and display stack element until a left parenthesis is encountered, but do not display it.
      iii. An operator: — if stack is empty or token has higher precedence than the element at TOS, push
      Note: "(" has the lowest precedence
      else pop and display the top stack element and repeat step # iii.
      iv. An Operand: — Display
3. Until the stack is empty, pop and display

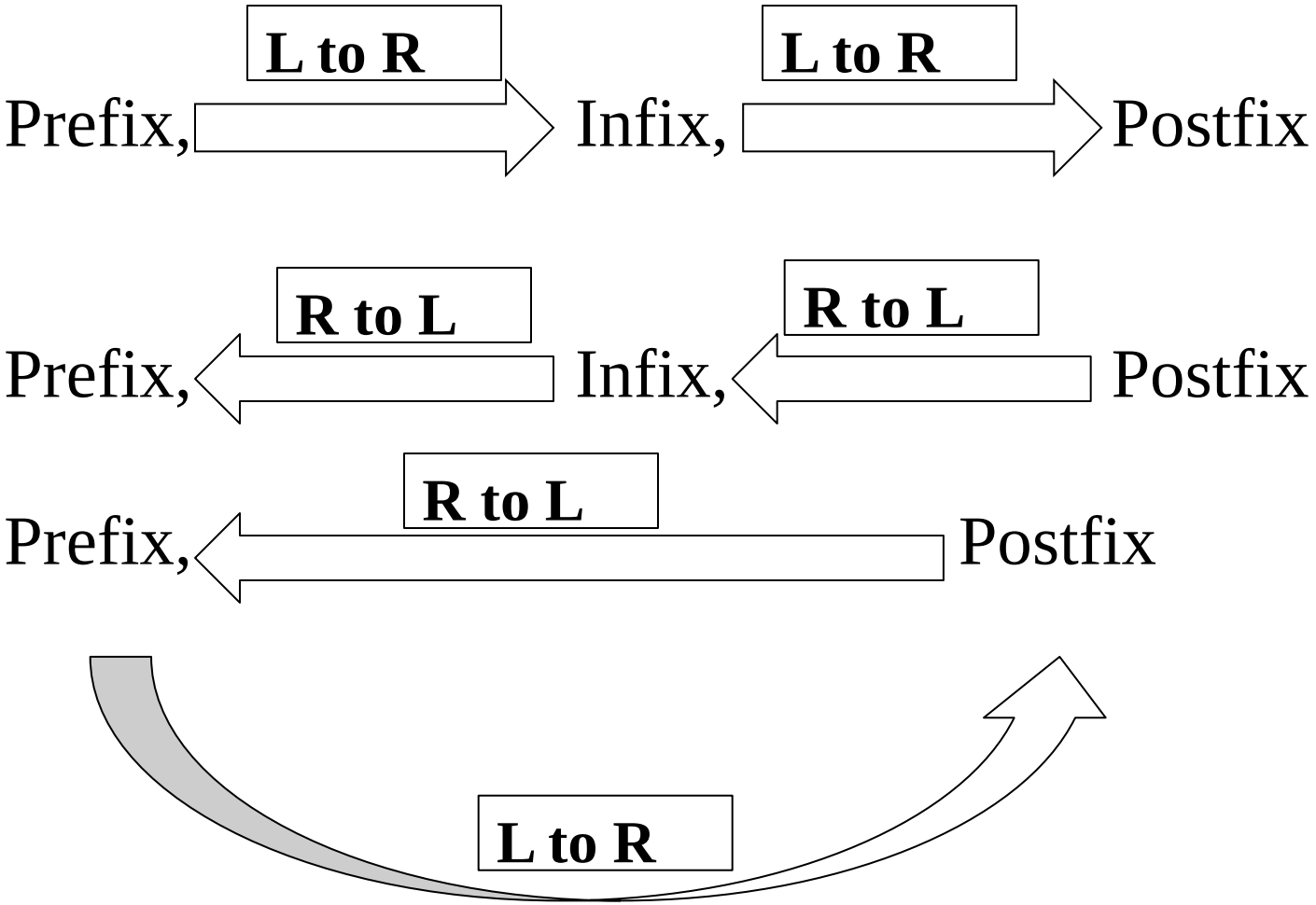
Converting Infix to RPN

(a+b)*(c+d) → ab+cd+*

Input Symbol	Stack	Remarks
(	(	Push
a	(	Operand – display – RPN → a
+	(+	Push as + has higher precedence than (
b	(+	Operand – display – RPN → a b
)		Pop till "(" is found and display – RPN → a b +
*	*	Push as stack is empty
(	* (	Push
c	* (	Operand – display – RPN → a b + c
+	* (+	Push as + has higher precedence than (
d	* (+	Operand – display – RPN → a b + c d
)	*	Pop till "(" is found and display – RPN → a b + c d +
End of input		Pop remaining symbols from the stack and display RPN → a b + c d + *

a+b*c/(d+e) → a b c * d e + / +

Input Symbol	Stack	Remarks
a		Operand – display – RPN → a
+	+	Push as stack is empty
b	+	Operand – display – RPN → a b
*	+ *	Push as * has higher precedence than +
c	+ *	Operand – display – RPN → a b c
/	+ /	Pop * and push / as * and / have the same precedence but / has higher precedence than + – RPN → a b c *
(	+ / (	Push
d	+ / (	Operand – display – RPN → a b c * d
+	+ / (+	Push as + has higher precedence than (
e	+ / (+	Operand – display – RPN → a b c * d e
)	+ /	Pop till "(" is found – RPN → a b c * d e +
End of input		Pop remaining symbols from the stack and display RPN → a b c * d e + / +


L to R
L to R
Prefix,
Infix,
Postfix
R to L
R to L
Prefix,
Infix,
Postfix
R to L
Prefix,
Postfix
L to R

Reading Materiel

- **Nell Dale – Chapter#4**
- **Schaum's Outlines – Chapter#6**
- **D. S. Malik – Chapter#7**
- **Introduction to Algorithm CLRS 3e Chapter # 10**
- **http://www.cs.man.ac.uk/~pjj/cs2121/fix.html**